Reducing Impact of Cache Miss Stalls in Embedded Systems by Extracting Guaranteed Independent Instructions

Garo Bournoutian
University of California, San Diego
9500 Gilman Dr. #0404
La Jolla, CA 92093-0404
garo@cs.ucsd.edu

Alex Orailoglu
University of California, San Diego
9500 Gilman Dr. #0404
La Jolla, CA 92093-0404
alex@cs.ucsd.edu

ABSTRACT

Today, embedded processors are expected to be able to run complex, algorithm-heavy, memory-intensive applications that were originally designed and coded for general-purpose processors. As such, the impact of memory latencies on the execution time increasingly becomes evident. All the while, it is also expected that embedded processors be power-conscientious as well as of minimal area impact. As a result, traditional methods for addressing performance and memory latencies, such as multiple issue, out-of-order execution and large, associative caches, are not aptly suited for the embedded domain due to the significant area and power overhead. This paper explores a novel approach to mitigating execution delays caused by memory latencies that would otherwise not be possible in a regular in-order, single-issue embedded processor without large, power-hungry constructs like a Reorder Buffer (ROB). The concept relies on both compile-time and run-time information to safely allow non-data-dependent instructions to continue executing while a memory stall has occurred. The simulation results show significant improvement in execution throughput of approximately 11%, while having a minimal impact on area overhead and power.

Categories and Subject Descriptors

B.8.0 [**Performance and Reliability**]: General; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*real-time and embedded systems*

CASES'09, Oct 11–16, 2009, Grenoble, France.

General Terms

Design, Performance

Keywords

embedded processors, data cache, pipeline stalls, compiler assisted hardware

1. INTRODUCTION

The prevalence of embedded processors in modern computational systems has grown significantly over the last few years. At the current rate, embedded processors will become increasingly ubiquitous throughout our society, resulting in a broader range of applications that will be expected to run on these devices. Even today, embedded processors are expected to be able to run algorithmically-complex, memory-intensive applications that were originally designed and coded for general-purpose processors. Furthermore, embedded processors are becoming increasingly complex to respond to this more diverse application base. Many embedded processors have begun to include features such as multi-level data caches, but the impact from larger memory access times occurring as a result of cache misses becomes increasingly evident [1].

With the constraints embodied by embedded processors, one typically is concerned with high performance, power efficiency, better execution determinism, and minimized area. Unfortunately, these characteristics are often adversarial, and focusing on improving one often results in worsening the others. For example, in order to increase performance, one adds a more complex cache hierarchy to exploit data locality, but introduces larger power consumption, more data access time indeterminism, and increased area. However, if an application is highly regular and contains an abundance of both spatial and temporal data locality, then the advantages in performance greatly out-

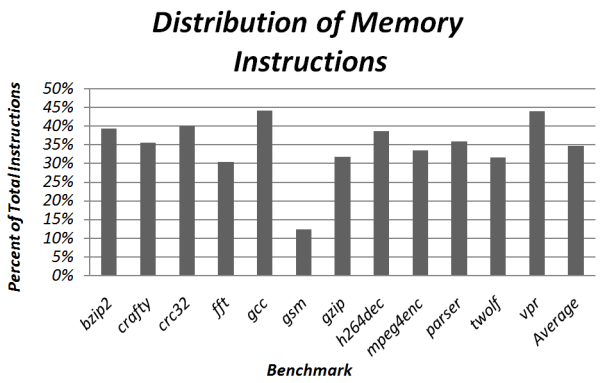

Figure 1: Average Distribution of Memory Instructions

weigh the drawbacks. On the other hand, as these applications become more complex and irregular, they are increasingly prone to thrashing. For example, video codecs, which are increasingly being included in wireless devices like cell phones, utilize large data buffers and significantly suffer from cache thrashing [2].

In particular, in embedded systems, where power and area efficiency are paramount, smaller, less-associative caches are chosen. Earlier researchers realized that these caches are more predisposed to thrashing, and proposed solutions such as the victim cache [3] or dynamically-associative caches [4] to improve cache hit rates. Yet, even with these various solutions to reduce cache misses, today's more aggressive and highly irregular applications inevitably will still suffer some cache misses.

Furthermore, embedded processors typically do not employ multiple-issue or out-of-order execution, as the area, power, and complexity impacts are prohibitive. Because of this single-issue behavior, when an instruction stalls the pipeline, subsequent instructions are not permitted to execute, even if those instructions are independent and ready for execution. A cache miss is an example of such a stalling instruction, and the processor must wait until the memory reference is resolved before allowing the pipeline to begin issuing instructions again. Each of these memory stalls can result in hundreds of processor cycles wasted in the event that the cache miss needs to read from main memory [5]. As shown in Figure 1, memory-based instructions account for approximately 35% of the total number of operations executed on average in applications. Thus, minimizing the impact of cache miss stalls can have a significant improvement on overall processor performance.

In this paper, we propose a solution that will utilize both the compiler and a nominal amount of specialized hardware to extract and allow guaranteed independent instructions to continue execution during memory stalls, which in turn reduces the overall execution time. In this manner, the delay induced by memory stalls can be effectively hidden. Fundamentally, this approach extends the communication link between compiler and processor architecture by transferring a small amount of application information directly to the microarchitecture without modifying the existent instruction set. Thus, by combining global application information known at compile-time with run-time cache behavior, we are able to intelligently keep feeding the pipeline with instructions we know are guaranteed to be independent of the memory operation. We show the implementation of this architecture and provide experimental data taken over a general sample of complex, real-world applications to show the benefits of such an approach. The simulation results show significant improvement in execution throughput of approximately 11% by hiding approximately 64% of cache miss stalls, while having a minimal impact on area overhead and power consumption.

2. RELATED WORK

In the last five years, the industrial embedded processor space has seen enormous expansion. Processors provided by companies such as ARM and Tensilica have become increasingly more powerful and complex, and are used in a wide variety of industrial applications. For example, current cell phone technology often incorporates both ARM9 and ARM11 embedded processors, along with a number of sophisticated specialized DSP processors, such as Qualcomm's QDSP6. These mobile phones are expected to handle a wide variety of purposes, from data communication to audio/video processing, and even GPS tracking. These target applications are becoming increasingly complex and memory-intensive, and numerous techniques have been proposed to address the memory access challenges involved. Unfortunately, embedded processors are often more highly constrained than general-purpose processors, and many general solutions are precluded by these tougher design constraints.

Various techniques have been proposed and used in the computer architecture community to attack the problem of memory stalls. In general, superscalar, out-of-order processors inherently can mitigate memory stalls by leveraging the reorder buffer (ROB) to dynamically allow multiple instructions to issue whenever resources are available and data

dependencies have been met [6, 7]. But, as mentioned in [8], even though in-order execution suffers from a 46% performance gap when compared to out-of-order, an out-of-order architecture is not ideal for embedded systems since it is too costly in terms of complexity and area.

As an alternative to costly out-of-order processors, [9] proposes having the compiler provide reordering by explicitly placing instructions into per-functional unit delay queues, which in turn still operate in an in-order fashion. While this is less complex than a full out-of-order processor, it still requires a rather large area overhead for the delay queues and does not allow for fine-grained, dynamic adjustment based on runtime cache hit/miss behavior.

Prefetching data into the cache is another well-established approach to reducing the performance impact of memory stalls. Software prefetching methods [10, 11, 12, 13] rely on the compiler to statically insert *prefetch* instructions before that actual load instruction. But, these instructions may cause code bloat and increase register pressure. Furthermore, purely software approaches cannot leverage runtime information relating to cache hit/miss behavior, and thus could result in wasted cycles from unnecessary prefetching. Alternatively, hardware prefetching methods [3, 14, 15, 16] utilize access patterns to predict cache misses and inject necessary prefetch logic. Unfortunately, as with all prediction schemes, these techniques heavily rely on memory access patterns and react poorly to applications with large or irregular memory accesses. Incorrect prediction can lead to cache pollution and a significant performance penalty.

[17] proposes using a combination of compiler and hardware support to prioritize instructions that are needed to keep the pipeline moving, and when none are available, allows for the buffered low-priority instructions to execute. While this proposed scheme is said to be for an in-order embedded processor, it unfortunately still relies upon a reorder buffer (ROB) and run-time register renaming, which are the largest contributors in area and power consumption in an out-of-order system. In fact, as exemplified by [18], the instruction issue queue of 56 entries in the HP PA-8000 utilizes 20% of the die area, which is impractical for most embedded systems.

3. MOTIVATION

A typical data-processing algorithm consists of data elements (usually part of an array or matrix) being manipulated within some looping construct. These data elements each effectively map to a predetermined row in the data cache. Unfortunately, different data elements may map to the same row due to the inherent design of caches. In this case, the data elements are said to be in "conflict". This is typically not a large concern if the conflicting data elements are accessed in disjoint algorithmic hot-spots, but if they happen to exist within the same hot-spot, each time one is brought into the cache, the other will be evicted, and this *thrashing* will continue for the entire hot-spot.

Given complex and data-intensive applications, the probability of multiple cache lines being active within a hot-spot, as well as the probability of those cache lines mapping to the same cache set, increases dramatically. As mentioned, much prior work has already gone into minimizing and avoiding cache conflicts and thrashing, but as applications continue to become more complex, the working set of data may exceed the capacity of the cache or lack localized regularity, both of which will degrade cache performance and lead to a larger miss rate. Additionally, since embedded processors are constrained to minimal area, power, and complexity, they often lack complex hardware to allow out-of-order execution (e.g. reorder buffer, register renaming, and multiple-issue). Thus, when these cache misses do occur, the pipeline must stall and wait until the missed cache line is resolved before continuing execution. Given the current trajectory of application complexity expected to run on embedded systems, the possibility of such cache misses and resulting memory pipeline stalls will become more prevalent and detrimental to execution speed.

To illustrate this point, Figure 2 shows an excerpt of assembly code from a GSM full-rate speech transcoder compression algorithm, which could be found on a typical GSM cellular phone. This basic block contains a single load instruction, and based on profiling, it was found that this particular instruction encountered a cache miss 4.07% of the time. If one were to assume that the cache miss penalty was 10 cycles and that all instructions take 1 cycle, the impact of cache misses would amount to an execution cycle increase of 5.09% for this

```
01: addiu  $18, $29, 16
02: addiu  $19, $0, 320
03: addiu  $17, $29, 336
04: lui    $20, 4096
05: addiu  $20, $20, 604
06: lw     $2, -32064($28)  # 4.07% miss
07: addiu  $4, $29, 16
08: jalr   $31, $2
```

Figure 2: GSM Example Basic Block

particular basic block. Since this particular basic block was executed 178,009 times, this would result in an additional 9,061 cycles, which translates to about 0.011ms of additional execution time overhead (assuming an 800MHz clock rate). Obviously, this example was looking at just a single basic block; a given application would have quite a number of basic blocks, each of which could suffer a similar fate and manifest into a much larger increase in execution time for the entire application.

The goal of this paper is to allow the compiler to analyze the basic block above, noting that the only instruction dependent on the memory load is the jump-and-link (*jalr*). Thus, the compiler can safely rearrange the code above to place the memory load at the beginning and leave the jump-and-link at the end, effectively positioning 6 instructions between the memory load and the next dependent instruction. This information is then provided to a specialized hardware structure when the application is loaded for execution. This hardware structure will then be able to detect if a cache miss occurred, and instead of blindly stalling the pipeline like before, it can safely know that 6 instructions can execute while the memory subsystem is resolving the cache miss. Thus, in this example, the execution cycle increase will become 2.04% instead of 5.09%, effectively hiding about 60% of the cache miss penalty.

4. IMPLEMENTATION

The proposed solution is composed of two parts: a *compile-time* mechanism, which will analyze and reorder the instructions within basic blocks and a *run-time* instruction execution mechanism that allows independent instructions to continue running during cache misses. Each part is described in detail in the following sections.

4.1 Compile-Time Analysis and Reordering

The first aspect of our methodology is to have the compiler statically analyze the application and identify those instructions that can safely be executed during a memory stall. Intuitively, these are the instructions that are not data dependent on a memory load and will not cause a hazard if executed prior to the completion of the memory load. Once these instructions are found, they will be aggregated and placed between the load instruction and any subsequent dependent instructions, which will enable us to execute those instructions during a cache miss.

In order to accomplish this analysis, we will leverage the *data flow graph* (DFG) normally used during compilation. The DFG provides an algorithmic representation of the data dependencies between a number of operations. To simplify and bound our analysis, we will operate on a *basic block* granularity, where a basic block is just a segment of code which has only one entry point and one exit point (e.g. no control flow changes exist within the block of code). Each basic block is represented with a DFG $G = (V, E)$, where each $v \in V$ is an instruction in the basic block. There exists a directed edge $e = (v_i, v_j) \in E$ if operation v_i is data dependent on operation v_j within the basic block.

We propose an algorithm that will make use of the DFG to analyze the instruction dependency characteristics and rearrange the instructions in a manner which will place memory loads earlier while pushing instructions dependent on these memory loads later within the basic block. This algorithm must fulfill the following constraints in order to maintain proper program execution and behavior:

- True data dependencies must be maintained
- The order of store operations must be maintained
- The order of load operations respective to store operations must be maintained

In particular, it is important to note that the constraints listed above with regard to memory store ordering could be relaxed if the effective destination address was statically guaranteed not to conflict with other memory load/store instructions. For simplicity, we conservatively assume that this cannot be guaranteed. However, our algorithm can easily be extended to support this additional compile-time analysis, which would increase the degrees of freedom available for reordering.

Keeping the above constraints in mind, our analysis algorithm performs the following steps to rearrange the instructions within a basic block:

1. Insert references for all load instructions into the "unvisited set"
2. Set the "top" pointer to point to the first instruction position in the basic block
3. Optionally remove any anti-dependencies via static register renaming[1] or immediate value adjustment
4. Select and remove the load instruction reference which depends on the *least* number of prior instructions from the "unvisited set"
5. Bubble the selected load instruction and those instructions it depends on as far towards the

[1]Requires a basic *register liveness* analysis, employing *use-define chains* to help detect write-after-read (WAR) dependencies.

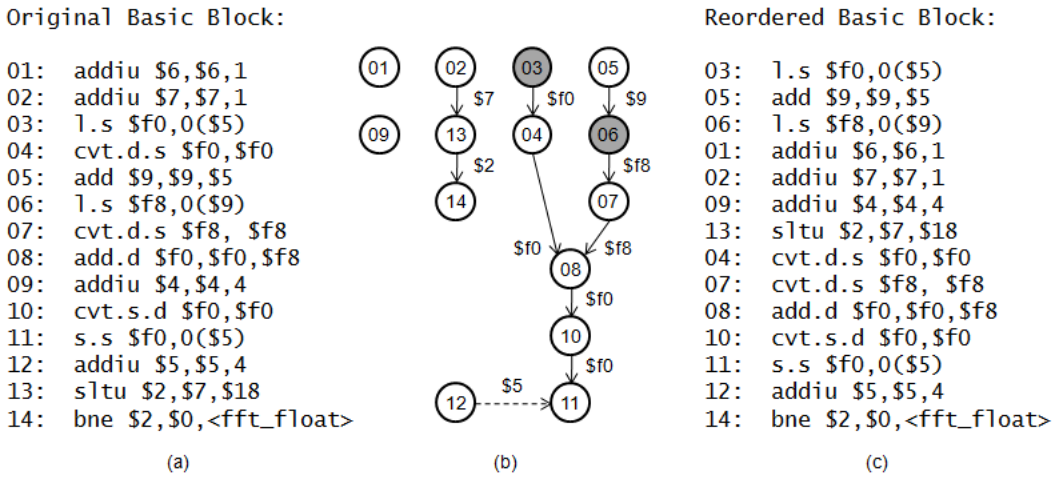

Figure 3: Excerpt from FFT algorithm with DFG and Reordering

"top" pointer position without violating constraints

6. Push the reference to the selected load instruction onto the "visited stack"
7. Set the "top" pointer to point to the position right after the new location of the selected load instruction
8. Repeat steps 4–7 until the "unvisited set" is empty
9. Set the "bottom" pointer to point to the last instruction position in the basic block
10. Move the "bottom" pointer to the second-to-last instruction position if the last position contains a control instruction
11. Pop a load instruction reference from the "visited stack"
12. Select the instruction within the "top" and "bottom" pointers which most directly depends on the selected load instruction reference
13. Bubble the selected instruction and those instructions that depend on it as far towards the "bottom" pointer position without violating constraints
14. Set the "bottom" pointer to point to the position right before the new location of the selected instruction
15. Repeat steps 12–14 until no more dependent instructions are found for the given load instruction reference or until the "top" and "bottom" pointers match
16. Repeat steps 11–15 until the "visited stack" is empty or until the "top" and "bottom" pointers match

After this algorithm completes, the basic blocks will have the following characteristics:

- Memory load instructions as early as possible
- Instructions that depend on memory loads as late as possible
- Instructions that depend on earlier memory loads will occur before instructions that depend on later memory loads
- Memory-independent instructions that can be moved will be placed in between the memory loads and instructions that depend on those loads

To illustrate this point, Figure 3(a) provides an example of a basic block from the *Fast Fourier Transform* algorithm, along with the corresponding DFG in Figure 3(b). Running the aforementioned algorithm will reorder the instructions as shown in Figure 3(c). Observe that instructions $01, 02, 09$, and 13 are located between the memory loads and memory-dependent instructions. Furthermore, instruction 04 is located before instruction 07. The importance of this second observation will be described shortly.

An additional observation relates to instruction 12. This instruction causes an anti-dependency (write-after-read) on instruction 11 (shown by the dotted line). In the above example, we did not eliminate this dependency. Static register renaming could remove this dependency, but at the cost of an additional instruction and added register pressure. But, an alternative in this instance is to note that instruction 11 ($s.s\ \$f0, 0(\$5)$) could be changed to ($s.s\ \$f0, -4(\$5)$) and then instruction 12 can be moved above it. This more aggressive anti-dependency analysis can help identify additional opportunities that assist in finding instructions that are memory-independent.

Now that the basic block instruction order has been primed, we will selectively annotate those ba-

sic blocks which have the greatest potential impact on run-time performance (e.g. the hot-spots). The data structure we will be creating is shown in Figure 4. Essentially, we need to capture the instruction address (PC) for the first load instruction of the basic block and the number of independent instructions as counted from that initial load instruction. Furthermore, we can capture the increase in independent instructions as we resolve each load instruction in the basic block (up to 9 in this example, including the initial load instruction).

The first entry in Figure 4 corresponds to the instructions shown in Figure 3(c) as an example of this data structure. As you can see, when the first load instruction (03) is encountered, we know that we can safely execute 6 additional instructions were a cache miss to occur. Furthermore, once instruction 03 completes, we can execute 1 more instruction, and when instruction 06 completes, we can execute 6 more instructions.

This compile-time data structure must then be stored and conveyed to the underlying hardware when the application is executed. To accomplish this, the data structure is encoded into the application binary's text segment at a special reserved address range. When a program binary is loaded onto the processor, this special address range will be read and the corresponding data structure populated with this information. The hardware mechanism that will detect and enable this independent instruction execution will be described in the next section.

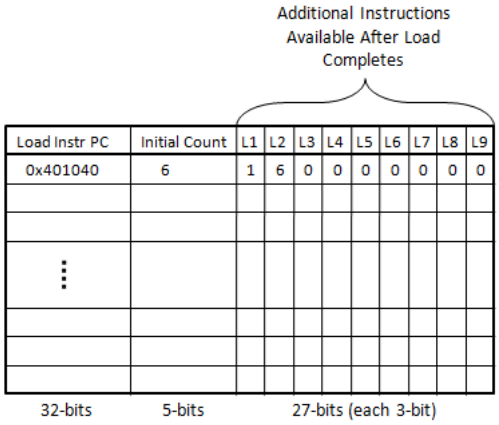

Figure 4: Basic Block Memory Independence Annotation (memTable)

4.2 Run-Time Stall Execution Mechanism

Now that the compile-time analysis has reordered the instructions and annotated the application's program binary, a hardware mechanism is needed to make use of this information. Intuitively, the goal of this mechanism is to leverage the annotated information provided at compile-time along with real-time information on cache misses and memory stall completion to allow the pipeline to keep executing when it would have otherwise needed to stall. To accomplish this behavior, we will need a nominal amount of additional hardware. It is important to note that the magnitude of this new hardware is greatly smaller than a typical out-of-order processor's reorder buffer (ROB).

The first structure we will need is the memory access table (*memTable*) that will hold the information stored in the application binary. This is essentially what is shown in Figure 4. Namely, this structure will hold the PC, initial count, and 9 additional increments for a given basic block's memory operations. A total entry width of 64-bits was chosen. This allows for 32-bits for the PC address, 5-bits for the initial instruction count, and 9 increment slots, each having 3-bits. The number of entries can be varied, but for the sake of this paper, we chose 256 entries, which will allow us to apply this optimization on up to 256 basic blocks within the application.

An additional feature of our implementation is that we allow subsequent load instructions to be "executed" during the stall of a previous load. As is typical with most embedded processors, the memory subsystem can only service one memory instruction at a given time. Thus, it is not possible to execute a load while a prior load is still waiting on a cache miss (e.g. this would be a structural hazard). But, instead of always assuming that a load may stall, which requires constraining the number of independent instructions found at compile-time to halt at the first memory instruction, we chose to take an innovative hardware approach, which will be described shortly. With respect to memory stores, we assume the use of a write-back buffer which can be serviced even while the memory subsystem is resolving a memory load.

The essential issue is that there *may* be a cache miss on a given load instruction at run-time, but then again, there may not be. Instead of always assuming the worst-case that is necessary at compile-time, we can leverage this run-time behavior to overcome this issue. We propose the addition of a small circular load instruction buffer (*loadBuf*). The purpose of this buffer is to accumulate the effective address, destination register, and memory-specific opcode for a load instruction when the memory subsystem is occupied. For example, if you were to have two back-to-back load instructions, the first instruction could hit or could miss.

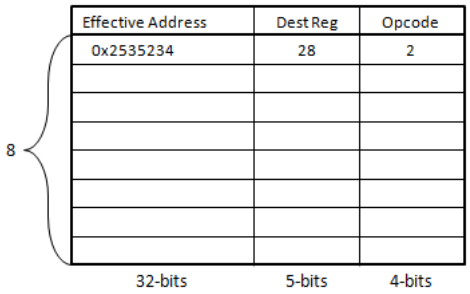

Figure 5: Load Instruction Buffer (loadBuf)

If it hits, the next instruction can execute normally. But, if the first instruction misses, we allow the second instruction to get executed in the pipeline and captured in this new load buffer. Immediately after the first instruction completes, the pipeline will inject the buffered memory signals to the memory subsystem and begin the second load. The structure of this load buffer is shown in Figure 5. We calculated the entry width to be 41-bits, to allow for the typical 32-bit address, 5-bit register number (either integer or floating-point), and 4-bit opcode (which can handle up to 15 types of memory load instructions). We chose 8 entries to correspond to the 9 additional increment positions in the *memTable* (since one of those positions is for the initial load instruction). Since this buffer is circular and has 8 entries, we would also need a 3-bit index register and 3-bits to keep track of how full the buffer is. Thus, this structure accounts for a total storage increase of 334 bits. This structure can be integrated into the memory subsystem to easily capture subsequent load instructions coming into the subsystem.

Lastly, we need to have a counter register (*counterReg*) to keep track of the number of instructions we executed and an increment register (*incrReg*) to store the additional increment values that may occur in the basic block. The *counterReg* lets us know if we have exhausted the number of guaranteed independent instructions and thus must stall if the memory access is not complete. This *counterReg* will be 7-bits, which will allow the initial 5-bit count, plus the possibility of 9 additional 3-bit increases without having an overflow. The *incrReg* is a shift register that will shift out the upper 3-bits and add them to the *counterReg* whenever a memory access completes.

The basic idea at run-time is to compare the PC at the memory stage with the entries in the *memTable*. If there is a match, load the corresponding count number and additional increments stored in the *memTable* into the *counterReg* and

```
In memory (M) stage:
  if( PC matches an entry in memTable ) {
    load memTable entry's count into counterReg
    load memTable entry's increment into incrReg
  }

  if( cacheMissSignal ) {
    if( counterReg > 0 ) {
      if( M stage instruction was load ) {
        latch memory instruction information inside subsystem
            (e.g. effective address and dest. reg)
        kill memory instruction (e.g. inject nop as result
            to WB stage)
      }
      if( EX stage instruction was a load ) {
        if( loadBuf is full ) {
          stall pipeline
        } else {
          push instruction to loadBuf and inject nop
        }
      }
    } else {
      stall pipeline (same behavior as default)
    }
  }
  else {
    if( loadInst just completed -- stall or no stall ) {
      if( M stage instr != instruction that just completed ) {
        create a bubble in M stage (e.g. only shift M & WB
            stages in pipeline)
        place memory data and dest reg as input to pipeline regs
      }
      take upper 3-bits of incrReg and add to counterReg
      left shift incrReg by 3-bits
    }
    if( loadBuf not empty ) {
      create a bubble in EX stage
      insert next load instruction as input to M stage
    }
  }

  if( counterReg > 0 ) {
    decrement counterReg
  }
```

Figure 6: Run-Time Independent Instruction Execution Behavioral Algorithm

the *incrReg*, respectively. If the memory access is a miss, continue executing and decrementing the *counterReg* until it becomes zero. At that point, you will need to stall the pipeline. If the cache miss resolves before the *counterReg* expires, the *incrReg*'s upper 3-bits are shifted out and added to the *counterReg*. This increase to the *counterReg* indicates the number of additional instructions that can now be executed since the previously-stalled memory reference completed. Additional logic exists to account for buffered loads. The full explanation of the run-time behavioral algorithm is provided in Figure 6.

An important observation is that this implementation will *never* result in worse cycle throughput than the original, non-optimized architecture. If one were to assume 100% cache hit rates, this implementation will behave identically with regard to instruction throughput when compared against the baseline architecture. Only in the event of cache misses, this implementation will allow additional

	Baseline			Our Design		
Benchmark	Total Cycles	Dead Cycles	Inefficiency	Dead Cycles	Inefficiency	Improvement
bzip2	2.33e11	3.71e10	15.94%	8.17e9	4.01%	12.43%
crafty	3.99e10	1.27e10	31.83%	3.16e9	10.41%	23.91%
crc32	5.52e8	1.81e7	3.29%	9.05e6	1.67%	1.65%
fft	6.73e8	4.34e7	6.45%	8.02e6	1.26%	5.26%
gcc	2.55e9	5.33e8	20.91%	1.78e8	8.12%	13.92%
gsm	6.53e8	1.66e7	2.54%	8.08e6	1.25%	1.30%
gzip	1.26e11	1.67e10	13.30%	3.99e9	3.54%	10.12%
h264dec	1.72e10	3.88e9	22.62%	1.47e9	9.94%	14.07%
mpeg4enc	1.16e10	1.07e9	9.23%	3.95e8	3.61%	5.83%
parser	1.75e10	4.07e9	23.26%	1.63e9	10.84%	13.93%
twolf	1.92e10	5.96e9	31.12%	3.95e9	23.04%	10.50%
vpr	1.57e10	4.96e9	31.67%	1.46e9	12.02%	22.34%

Table 1: Overall Execution Run-Time Improvement

instructions to be executed in otherwise dead cycles during the memory stall. Furthermore, it is important that we utilize a run-time control for the algorithm, since having multi-level caches in our architecture can lead to having dynamic miss penalties that cannot be assumed at compile-time alone.

5. EXPERIMENTAL RESULTS

In order to assess the benefit from this proposed architectural design, we utilized the SimpleScalar toolset [19]. We chose a representative embedded system configuration, having a 256-set, direct-mapped L1 data cache with a 32-byte line size. This processor model utilized a typical in-order simulation engine, with dedicated L1 caches for data and instruction memory, backed by a unified L2 cache.

Twelve representative benchmark programs from the SPEC CPU2000 suite [20], the MediaBench video suite [21], and the MiBench telecomm suite [22] are used: *bzip2* - compression program; *crafty* - high-performance chess playing application; *crc32* - 32-bit CRC framing checksum; *fft* - discrete fast Fourier transform program; *gcc* - C compiler; *gsm* - telecomm speech transcoder compression program; *gzip* - LZ77 compression program; *h264dec* - H.264 video decoder; *mpeg4enc* - MPEG-4 video encoder; *parser* - word processing application; *twolf* - CAD place-and-route simulation; and *vpr* - FPGA place-and-routing.

Table 1 provides the total number of execution cycles, dead execution cycles from cache miss stalls, percentage of execution overhead due to memory stalls, the number of dead cycles after our architecture allowed independent instructions to execute during memory stalls, our percentage of execution overhead due to memory stalls, and our percentage improvement over the baseline in terms of overall run-time execution. In addition, Figure 7 graphically shows the percent improvement in run-time execution behavior using our implementation. The average of the execution time improvement across the benchmarks for our proposed implementation is 11.27%.

Additionally, Figure 8 shows a metric of how well our algorithm was able to identify and allow independent instructions to fill in the delays caused by memory stalls. As one can see, a large amount of the stalled cycles are able to be utilized using our implementation. The average quantity of dead stall cycles that were able to be recouped to run useful instructions was 64.02%, which is rather substantial.

With regard to power efficiency, since we only use a nominal amount of additional hardware, the impact of the proposed technique is quite minimal. Overall, the structures proposed account for approximately 2kB of additional storage elements, along with some necessary routing signals and muxing. Since our implementation does not

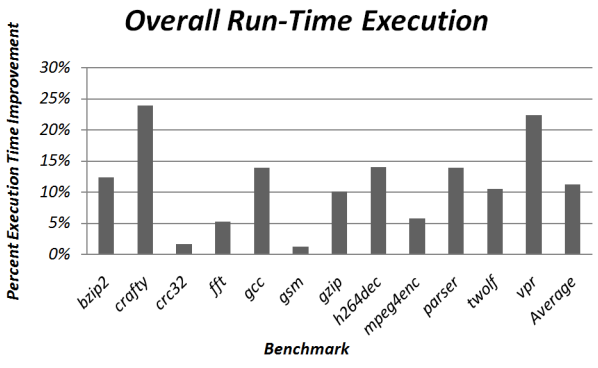

Figure 7: Overall Run-Time Execution Improvement

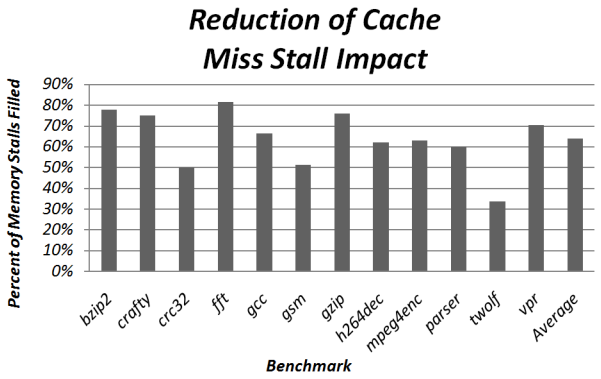

Figure 8: Success of Filling Stalls With Independent Instructions

rely on creating some central register/operand storage area, we do not suffer the same routing and muxing overhead that exponentially grows when dealing with reorder buffers and instruction queues. Furthermore, indexing into our hardware structure only occurs during load instructions, as opposed to every single instruction. Thus, when compared to the complexity of a fully dynamic instruction reordering processor, our implementation is far more efficient. According to [23], a typical out-of-order processor's ROB, register renaming table, and instruction queue on average consume about 27%, 13.5%, and 26% of the total energy usage of the processor, respectively. This combines for a total energy consumption of about 67% of the total processor, a figure that is not feasible when considering the embedded domain. On the other hand, the energy consumption of our proposal is about 0.6% of the total energy footprint for a comparable in-order processor, using a CACTI [24] approximation. Therefore, our proposed technique can deliver most of the same benefits with regard to resolving memory stalls, but at a fraction of the energy consumption and area utilization of an out-of-order processor.

6. CONCLUSIONS

We have presented a novel architecture for leveraging static compile-time analysis information and hiding the impact of memory stalls by configuring and allowing independent instructions to continue executing during a memory stall at run-time. As shown, memory instructions account for a large proportion of the instructions within an application. Although much work has gone into mitigating cache misses, processors still encounter a substantial amount of overhead when misses do occur. Embedded processors, being far more constrained in terms of power consumption and area constraints, cannot easily leverage the same technology used in general-purpose machines to overcome memory stalls. In particular, out-of-order execution is prohibitively expensive in terms of area and power utilization in the embedded domain.

By leveraging both compile-time and run-time information, we were able to propose an architecture capable of delivering some of the benefits of out-of-order execution, but at a far less cost. The achievement of these goals has been confirmed by extensive experimental results. A significant reduction in the number of dead cycles due to memory stalls has been demonstrated by a representative set of simulation results. The proposed technique has significant implications for embedded processors, especially with regard to high-performance, power-sensitive devices such as cell phones and MP3 players, as it yields improvements to execution time while minimally impacting power consumption.

As embedded processors continue to spread and become ubiquitous, it is essential to maintain high performance, low power, and small size. The proposed architecture fulfills these requirements and enables embedded processors to continue to mature and be able to handle exceedingly complex and aggressive applications.

7. REFERENCES

[1] Maurice V. Wilkes. The memory gap and the future of high performance memories. *SIGARCH Computer Architecture News*, pages 2–7, 2001.

[2] Li Lee, Srikanth Kannan, and Jose Fridman. MPEG4 video codec on a wireless handset baseband system. In *Proc. Workshop Media and Signal Processors for Embedded Systems and SoCs*, 2004.

[3] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Computer Architecture News*, pages 364–373, 1990.

[4] Garo Bournoutian and Alex Orailoglu. Miss reduction in embedded processors through dynamic, power-friendly cache design. In *DAC '08: Proceedings of the 45th Annual Conference on Design Automation*, pages 304–309, New York, NY, USA, 2008. ACM.

[5] Eric Sprangle and Doug Carmean. Increasing processor performance by implementing deeper pipelines. *SIGARCH Computer Architecture News*, pages 25–34, 2002.

[6] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM*

Journal of Research and Development, pages 25–33, 1967.

[7] James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *ISCA '85: Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[8] Sébastien Hily and André Seznec. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pages 64–67, Washington, DC, USA, 1999. IEEE Computer Society.

[9] J.P. Grossman. Cheap out-of-order execution using delayed issue. In *ICCD '00: Proceedings of the 2000 IEEE International Conference on Computer Design*, pages 549–551, 2000.

[10] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *ASPLOS-IV: Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, New York, NY, USA, 1991. ACM.

[11] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. *SIGARCH Computer Architecture News*, pages 43–53, 1991.

[12] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-V: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, New York, NY, USA, 1992. ACM.

[13] Abdel-Hameed A. Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In *ICS '01: Proceedings of the 15th International Conference on Supercomputing*, pages 486–500, New York, NY, USA, 2001. ACM.

[14] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 176–186, New York, NY, USA, 1991. ACM.

[15] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *MICRO 25: Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 102–110, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[16] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, New York, NY, USA, 1997. ACM.

[17] Sanghyun Park, Aviral Shrivastava, and Yunheung Paek. Hiding cache miss penalty using priority-based execution for embedded processors. In *DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1190–1195, 2008.

[18] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGOPS Operating Systems Review*, pages 2–11, 1996.

[19] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, pages 59–67, 2002.

[20] SPEC CPU2000 Benchmarks. http://www.spec.org/cpu/.

[21] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[22] 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. In *WWC '01: Proceedings of the IEEE International Workshop on Workload Characterization*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[23] Daniele Folegnani and Antonio González. Energy-effective issue logic. *SIGARCH Computer Architecture News*, pages 230–239, 2001.

[24] Steven J. E. Wilton and Norman P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal on Solid-State Circuits*, 31(5):677–688, 1996.